Lidando com Transações

Site: Moodle institucional da UTFPR
Curso: CETEJ34 - Sistemas de Banco de Dados - JAVA_XXX (2024_01)
Livro: Lidando com Transações

Impresso por: PAULO ROBERTO DOS SANTOS
Data: terça-feira, 16 jul. 2024, 17:36

Índice

1. Lidando com Transações

Uma das características do Spring Data é a utilização do controle de transações. No bloco 2, onde foi apresentada a classe de configuração do Spring Data JPA, foram declarados dois recursos com essa finalidade, a anotação **@EnableTransactionManagement** para habilitar o controle transacional e o método **transactionManager()**.

Por padrão, a interface **CrudRepository** faz uso de controle transacional, assim, seus métodos e também as interfaces que a estendem, vão lidar com este recurso.

Isso significa, que todos os métodos herdados e assinados em **ContatoRepository** e **EnderecoRepository** são também transacionais.

Esta informação é importante porque o controle de transações de **CrudRepository** possui uma configuração que seta os métodos apenas para leitura. Ou seja, fora os métodos de escrita que são herdados, todos os outros, inclusive os que você adiciona nas suas interfaces serão apenas habilitados para leitura.

Então, se desejar criar um método de escrita, como um update, você vai precisar sobrescrever o controle transacional original.

Um exemplo disso são os métodos **updateCidadeByid()** e **deleteEndereco()** apresentados na **Listagem 4.22**.

Caso você tente executá-los, uma exceção do tipo **TransactionRequiredException** seria lançada.

Isto porque, como já dito, qualquer novo método assinado nas interfaces será por padrão configurado como uma operação de leitura.

Para modificar essa característica é preciso usar uma anotação do tipo **@Transactional**, alterando a transação de leitura para escrita. Mas antes de ver como isso é feito, vamos conhecer um pouco mais dessa anotação e suas propriedades.

- @Transactional · se incluída sobre a assinatura de uma classe ou interface gera uma configuração padrão entre todos os métodos desta classe ou interface. Mas pode também ser incluída individualmente sobre a assinatura de cada método, tornando assim, cada configuração restrita ao método anotado;
- readOnly· é uma propriedade da anotação que configura a transação para leitura ou escrita. Por padrão, na anotação, seu valor é false. Ou seja, a transação será para leitura e escrita. Caso seja true, a transação será apenas para leitura;
- propagation - esta propriedade possibilita a opção de especificar o comportamento do evento, em que um método transacional é executado, quando um contexto de transação já existe. Por exemplo, o código pode continuar a ser executado na transação existente, ou a transação existente pode ser suspensa e uma nova será criada;
- isolation - esta propriedade especifica o grau em que uma transação é isolada de outras transações. O padrão é o nível de isolamento do SGBD;
- timeout - é uma proprieda de que define o tempo limite que uma transação continuará ativa antes de sofrer um rollback. Por padrão, o tempo limite é o da infraestrutura acessada (SGBD);
- trollBackFor - esta propriedade serve para definir as classes de exceções que vão lançar a exceção em uma situação de rollback. As classes devem ser subclasses de **Throwable**. Se a propriedade não for usada, o padrão será **RuntimeException** e **Error**;

As principais propriedades de **@Transactional** são estas que foram listadas. Porém, algum as delas possuem uma lista própria de valores que definem qual característica devem assumir na transação.

Entre elas, a **propagation** e a **isolation**. Para conhecer algumas dessas características ligadas à propagação, confira a tabela 5.1.

Propagation	Objetivo
REQUIRED	O método participa da transação atual, caso não exista uma transação, uma nova será criada. Este é o valor padrão da **@Transactional**.
SUPPORTS	O método participa da transação atual, se não existir uma transação, ele é executado sem qualquer transação.
MANDATORY	O método participa da transação atual, se nenhuma existir, uma exceção será lançada.
REQUIRES_NEW	O método é executado em uma nova transação. Se uma já existe, ela será suspensa.
NOT_SUPPORTED	O método executa sem nenhuma transação. Se uma transação existir, ela será suspensa.
NEVER	O método é executado sem uma transação. Se uma existir, uma exceção será lançada.
NESTED	O método é executado dentro de uma transação aninhada se uma transação já existe. Caso contrário, ela assume o comportamento de REQUIRED.

TABELA 5.1: TIPOS DE PROPAGAÇÃO CONFIGURÁVEIS NO SPRING DATA.

Para saber quais são os valores referentes às características ligadas ao isolamento de transações, confira a Tabela 5.2.

Isolation	Objetivo
DEFAULT	Assume o nível de isolamento do gerenciador de dados. É o valor padrão de **@Transactional**.

Isolation	Objetivo
READ_UNCOMMITTED	É o nível de menor isolamento, permitindo a leitura antes da confirmação. Ou seja, uma segunda transação pode recuperar os dados que estão sendo processados por outra transação antes mesmo dela ser finalizada. Não é recomendado na prática.
READ_COMMITTED	Este nível apenas proíbe uma transação de ler uma linha com alterações não confirmadas.
REPEATABLE_READ	Este nível isola o conjunto de dados lidos de uma transação. Não permitindo a leitura de dados alterados ou excluídos, mesmo que confirmados pela transação corrente. Porém ele permite a leitura de novos registros confirmados por outras transações.
SERIALIZABLE	É o maior nível de isolamento. Não permite que nenhuma outra transação acesse os dados da transação atual.

TABELA 5.2: NÍVEIS DE ISOLAMENTO EM TRANSAÇÕES CONFIGURÁVEIS NO SPRING DATA.

1.1. Sobrescrevendo o Controle Transacional Padrão

Após a anotação **@Transactional** e suas propriedades terem sido apresentadas, vamos abordar o padrão transacional fornecido pela interface **CrudRepository**.

Se você abrir o código-fonte dessa interface, não verá nenhuma anotação referente a controle de transações. Neste caso, o controle é realizado diretamente por outras classes ou interfaces que fazem parte do Spring Data.

Mas, caso uma anotação estivesse explicitamente declarada em **CrudRepository** seria algo mais ou menos assim:

```
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public interface CrudRepository<T, ID extends Serializable>
   extends Repository<T, ID> {
   // código omitido nesta listagem
}
```

Neste exemplo, foram destacadas apenas as propriedades **readOnly** e **propagation**, todas as demais são configuradas com o valor padrão referente a cada propriedade já apresentada anteriormente.

Veja que embora a **readOnly** tenha por padrão da anotação o valor **false**, ou seja, para leitura e escrita, a interface **CrudRepositoty** configurou este valor como **true**, limitando os métodos apenas para leitura.

Internamente, os métodos de escrita desta interface sobrescrevem o valor true para false. Assim, os métodos de escrita herdados podem ser usados normalmente para esta operação, mas quando você criar os seus próprios métodos de escrita, vai precisar sobrescrever o valor padrão de **CrudRepository**.

Portanto, para sobrescrever o valor padrão de qualquer propriedade já definida por **CrudRepository**, basta incluir a anotação diretamente nos métodos de sua interface que precisa sofrer a alteração no processo transacional.

Confira um breve exemplo de como proceder com esta alteração no método **deleteEndereco()** da interface **EnderecoRepository**:

```
@Transactional(readOnly = false)
@Modifying
@Query("delete from Endereco e  where e.id = ?1")
int deleteEndereco(Long id);
```

Com a inclusão de @Transaction e sua propriedade **readOnly** como false, tornamos o método **deleteEndereco()** perfeitamente executável para escrita, sem que a exceção citada anteriormente seja lançada.

Entretanto, o controle transacional não precisa ser adicionado diretamente dentro da interface de repositório. O ideal é que isto seja realizado em outra camada, que não a de persistência, como a camada de serviço.

Isto porque, quando se inclui esse controle no repositório, você está definindo uma transação exclusiva para aquele método. Se outro método for executado, ele não vai compartilhar essa transação. E em alguns casos é uma boa ideia compartilhar a transação entre a execução de diferentes métodos.

Então, observe a seguinte situação em que o controle transacional é distinto para uma única operação que trabalha com dois métodos do repositório. Para isso, observe o código apresentado na Listagem 5.1.

LISTAGEM 5.1: EXECUTANDO DOIS MÉTODOS COM DIFERENTES TRANSAÇÕES.

```
@Service
public class EnderecoService{
   @Autowired
   private EnderecoRepository enderecoRepository;
   @Autowired
   private ContatoRepository contatoRepository;
   public void salvar(Endereco endereco, Contato contato) {
      enderecoRepository.save(endereco);
      contatoRepository.save(contato);
   }
}
```

Analisando o código da classe **EnderecoService** se pode notar que existem dois *beans* injetados: **EnderecoRepository e ContatoRepository**. A operação a ser realizada está no método **salvar()**.

Veja que este método recebe dois objetos como argumentos, um com os dados do endereço e outro com os dados do contato.

Quando a tabela **Contatos** foi criada no banco de dados, a coluna de chave estrangeira que representa **Enderecos** foi incluída com a instrução not null. Assim, para salvar um contato, obrigatoriamente deve ter um endereço.

No método **salvar()** existe a chamada a dois outros métodos e cada um deles vai usar seu próprio controle de transação. Isto porque, este controle está definido nos repositórios de **Endereco** e de **Contato**.

Então, o processo será o seguinte:

1. A ação **enderecoRepository.save()** vai usar uma transação exclusiva, que vai salvar o endereço no banco de dados e confirmar o sucesso da transação;
2. A ação **contatoRepository.save()** vai usar uma transação exclusiva, que vai tentar salvar a entidade **contato**. Como o objeto contato não contém um **endereco**, uma exceção de integridade será lançada pelo banco de dados e a operação não será confirmada. Assim, o contato não será salvo na tabela.

Após essas duas ações, o resultado será: um endereço salvo no banco de dados, sem nenhum contato vinculado a esse endereço.

Agora, vamos analisar um novo cenário, onde o controle transacional é único para os métodos **enderecoRepository.save()** e **contatoRepository.save()**.

Para isso, será necessário adicionar a anotação de transação no topo do método **salvar()** de **EnderecoService**, como mostra a Listagem 5.2.

LISTAGEM 5.2: EXECUTANDO DOIS MÉTODOS EM UM MESMO CONTROLE DE TRANSAÇÃO.

```
@Service
public class EnderecoService{
   @Autowired
   private EnderecoRepository enderecoRepositoyr;
   @Autowired
   private ContatoService conttaoRepository;
   @Transactional(readOnly = false)
   public void salvar(Endereco endereco, Contato contato) {
      enderecoRepository.save(endereco);
      contatoRepository.save(contato);
   }
}
```

Usando este novo cenário, os processos decorrentes desta operação seriam os seguintes:

1. O método **salvar()** de **EnderecoService** vai abrir uma transação;
2. A ação **enderecoRepository.save()** vai usar a transação existente e salvar o endereço no banco de dados, sem executar a confirmação (commit);
3. A ação **contatoRepository.save()** vai usar a transação existente e tentará salvar a entidade contato. Como este objeto **contato** não contém um **endereco**, uma exceção de integridade será lançada;
4. Como uma exceção foi lançada dentro da transação, um *rollback* será executado em todas as operações da transação. Desta forma, a operação do passo 2 não sofrerá a confirmação, assim como, a do passo 3;

Analisando esses quatro passos apresentados, é possível deduzir que um endereço não foi salvo no banco de dados devido ao *rollback*. Diferentemente do que aconteceu no processo em que cada operação tinha seu próprio controle transacional.

Por isso, acaba sendo mais vantajoso trabalhar com o controle de transações fora dos repositórios. Fica mais fácil controlar as operações que precisam ser executadas no contexto em que uma depende do sucesso de outra.

Agora que o processo de controle de transações foi explicado, para evitar que uma exceção venha a ocorrer no método **salvar()**, basta proceder como no exemplo a seguir:

```
@Transactional(readOnly = false)
public void salvar(Endereco endereco, Contato contato){
   enderecoRepository.save(endereco);
   contato.setEndereco(endereco);
   contatoRepositroy.save(contato)
}
```

Assim, a exceção de integridade não seria lançada por conta da não existência de um objeto **endereco** ao salvar a entidade **contato**.